V-COLLIDE: Accelerated Collision Detection for VRML*

Thomas C. Hudson Ming C. Lin† Jonathan Cohen Stefan Gottschalk Dinesh Manocha
Department of Computer Science
University of North Carolina‡
http://www.cs.unc.edu/~geom/collide.html

Abstract

Collision detection is essential for many applications involving simulation, behavior and animation. However, it has been regarded as a computationally demanding task and is often treated as an advanced feature. Most commonly used commercial CAD/CAM packages and high performance graphics libraries, such as SGI Performer, provide limited support for collision detection. As users continue to stretch the capabilities of VRML, collision detection will soon become an indispensable capability for many applications. In this paper, we present a system for accelerated and robust collision detection and describe its interface to VRML browsers. We demonstrate that it is possible to perform accurate collision detection at interactive rates in VRML environments composed of large numbers of complex moving objects.

CR Categories and Subject Descriptors: D.3.2 **[Programming Languages]:** Language/Classifications — Virtual Reality Modeling Language 2.0; I.3.1 **[Computer Graphics]:** Graphics Systems — Distributed/Network Graphics; I.5.3 **[Computer Graphics]:** Computational Geometry and Object Modeling

Additional Key Words and Phrases: Virtual Reality Modeling Language (VRML), collision detection

1 Introduction

The VRML 2.0 specification calls for collision detection to be performed between a volume surrounding an avatar's viewpoint and the scene geometry. However, this is only a small subset of the collision detection functionality useful to VRML applications. In addition to aiding navigation, collision detection is pivotal to simulating physics in virtual environments or avatar behaviors in cyberspace [8, 13]. Applications currently under investigation, such as collaborative design of CAD/CAM models in distributed virtual environments, require a realistic simulation that couples collision detection and dynamic response.

The problem of collision detection has been explored in the literature of computer graphics, robotics, computational geometry, computer animation, and physically-based modeling. Numerous approaches based on bounding boxes, spatial partioning, geometric reasoning, numerical methods, and analytical methods have been proposed [3, 6, 7]. However, none of these algorithms or systems satisfies the demanding requirements of general-purpose collision detection in VRML browsers.

Main Contribution: In this paper, we present a system (V-COLLIDE) for interactive collision detection among arbitrary polygonal models undergoing rigid motion in VRML environments. We unify several techniques from previous work in large-scale collision detection and hierarchical data structures [4, 5], and propose a clean integration of the resulting libraries with VRML 2.0 [2]. Our system offers a practical toolkit for performing interactive and robust collision detection in VRML environments.

Organization: In Section 2 of this paper, we describe the desired characteristics of a collision detection system for VRML applications. Section 3 presents the overall system architecture, drawing from previous methods for collision detection. We address the interface necessary between the scene graph and the collision detection system. Section 4 discusses our prototype implementation. We specify a simple yet complete interface between a browser's internals and the collision detection library. We report our preliminary results and analyze the performance of the system. The paper concludes in section 5 with directions for possible future work, taking into consideration the evolution we expect to see in the use of VRML.

*Supported in part by a Sloan fellowship, ARO Contract P-34982-MA, NSF grant CCR-9319957, NSF grant CCR-9625217, ONR contract N00014-94-1-0738, ARPA contract DABT63-93-C-0048, NSF/ARPA Science and Technology Center for Computer Graphics & Scientific Visualization NSF Prime contract No. 8920219.

†Also with the U.S. Army Research Office.

‡Sitterson Hall, University of North Carolina, Chapel Hill, NC 27599-3175. {hudson,lin,cohenj,gottscha,manocha}@cs.unc.edu

2 Desiderata

Given the performance criteria (speed, functionality and library interface) imposed by VRML environments, a collision detection system must be:

- **Dynamic** – A dynamic scene graph is the core of VRML 2.0. Not only will the user's viewpoint move through the scene, but objects will move, and may appear or disappear at any time. The data structures for collision detection need to take this dynamic behavior into account. Efficient maintenance of spatial data structures in dynamic environments is still an open research topic.
- **Interactive** – VRML is an interactive environment demanding a high frame rate. At a minimum the browser must perform collision detection between the avatar and its surrounding environment, but it would also be advantageous to detect collisions between arbitrary objects in the scene. Most approaches in the literature lack the robustness or the real-time performance required by VRML applications.
- **General** – A scene will contain objects of arbitrary topology. We should avoid assumptions about object motion (bounds on velocity or acceleration, predefined trajectories), the geometry of objects (convexity, solids, manifolds, other topological constraints), or the richness and correctness of data structures (winged-edge representations or "clean" geometry free of degeneracies). The system itself should be useful for applications not yet conceived, extendible to distributed multiuser simulation, and portable.

3 System Architecture

In this section we describe the architecture of V-COLLIDE and propose a method for the VRML scene graph to control the collision detection library.

3.1 Hierarchical Approach

Our proposal takes a multi-level approach to the problem of collision detection, similar to that used in the I-COLLIDE library [4]. A quick conservative approximation finds potentially-colliding pairs of objects among the entire database (using the n-body sweep-and-prune algorithm from I-COLLIDE), after which a pairwise test taken from RAPID [5] determines whether two objects marked as overlapping actually collided. Figure 1 shows the architecture of our collision detection library, V-COLLIDE.

The first level of V-COLLIDE computes minimal *axis-aligned bounding boxes* (AABBs) for every object in the scene. The endpoints of these boxes are sorted into three lists, one for each coordinate axis. As objects move, these lists must be re-sorted on every frame. We use insertion sort to take advantage of the expected frame-to-frame coherence in object positions. The complexity of performing this sorting is thus proportional only to the number of moving objects and the density of the neighborhoods through which they move. Only pairs of objects whose bounding boxes overlap in all three dimensions are passed to the pairwise test module of the system.

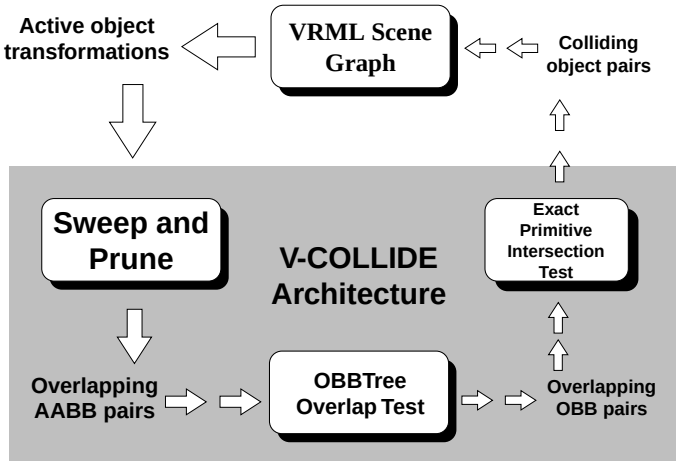

Figure 1: *The system architecture of V-COLLIDE.*

The next level detects pairwise collisions. Our implementation computes a tree of *oriented bounding boxes* (OBBs) for every object, with a box containing the entire object as the root and boxes only containing one or a very few primitives as the leaves. To check for collision between a pair of objects, we can descend their OBB hierarchies to find any leaf boxes which overlap, and then perform exact intersection tests between the triangles in the overlapping leaves. For efficient overlap tests of the boxes, we use the separating axis theorem [5].

We use the OBB construction method from RAPID. A top-down recursive approach partitions the primitives in a box into two sub-boxes, based on the location of their centers. This partitioning is heuristic, is a lightweight approach suitable for online computation, and gives reasonable results for collision detection.

Maintaining spatial hierarchies or partitions over complex, dynamic data is still an open research topic. Rather than attempting to extend the OBB tree algorithms to handle dynamic data, we concentrate on making OBB tree construction sufficiently fast that when the contents of a medium-sized tree change we can afford to destroy and rebuild the tree.

3.2 VRML 2.0 Interface

Collision detection normally deals with geometric contacts between two distinct objects. The geometric primitives that make up a single object are fixed in reference to one an-

other, and should not be tested for collision with one another. However, the scene graph of a VRML file does not give us any such differentiating information. It is impossible to tell how a Group node's children are related: by belonging to the same object, by inheriting the same transform for purposes of efficiency, or by some semantic criterion irrelevant to rendering. We propose to use the type extensibility of VRML 2.0 (the PROTO and EXTERNPROTO constructs) to allow a scene graph to specify to the browser where these divisions between the objects are.

We define the node type CollisionObject; it is similar to the Collision node in the VRML 2.0 spec, but informs the browser that the child geometry is a single object. The CollisionObject has an extra eventOut, named *collideObjects*, that signals which objects it hit at *collideTime*. The CollisionObject also has extra eventIns, *addIgnoredObjects* and *removeIgnoredObjects*, and an associated exposedField, *ignoredObjects*. These permit fine control over which pairs of objects are tested for collisions: a node whose *collide* field is TRUE defaults to testing against all other objects for collisions, but will ignore those listed in its *ignoredObjects* field.

The URN specification in our EXTERNPROTO (Figure 2) is a network-wide unique string following the naming conventions of the URN namespace[12]. It indicates unambiguously to a browser with a conforming collision detection implementation that this is our CollisionObject node, version 1. For browsers that do not recognize the URN, the alternate implementation of CollisionObject (Figure 3) is as a Collision node, with the extra fields and events ignored.

The drawback to embedding this information in the VRML file, is that using the *ignoredObjects* field requires some awkward circumlocutions, shown in Figure 4. VRML has no facility for forward references, and since we may have n objects which need to reference one another, it becomes necessary to nest their declarations up to n levels deep. The alternative is to use the initialize() method of a Java script or equivalent functionality to set up the *ignoredObjects* fields when the file is loaded.

```
EXTERNPROTO CollisionObject
  [ eventIn MFNode addChildren
    eventIn MFNode removeChildren
    eventIn MFNode addIgnoredObjects
    eventIn MFNode removeIgnoredObjects
    exposedField MFNode children
    exposedField MFNode ignoredObjects
    exposedField SFBool collide
    field SFVec3f bboxCenter
    field SFVec3f bboxSize
    field SFNode proxy
    eventOut SFTime collideTime
    eventOut MFNode collideObjects ]
  { "URN:edu.unc.cs:geom:CollisionObject:1"
    "http://www.cs.unc.edu/~geom/
     V_COLLIDE/CollisionObject.wrl" }
```

Figure 2: *EXTERNPROTO for CollisionObject*

4 Implementation

We have developed the V-COLLIDE library and designed an interface from the library to VRML browsers. A simple viewing program – not a full-fledged VRML 2.0 browser – has been used to test the functionality and performance of our implementation.

4.1 External Interface

Our library interfaces to the browser with a simple API, tailored to expected VRML requirements. This API is given in Table 1. Each object is added to the collision detection database by calling col_create_object(), col_add_triangle() for each triangle contained in a tessellation, and then col_finish_object(). These objects may be CollisionObjects specified in the scene graph, subobjects created as on-the-fly optimizations by the browser to spatially partition the scene geometry, or the radius around the avatar specified by the currently bound NavigationInfo to detect avatar collisions with the scene.

Calling col_finish_object() builds (or rebuilds) the OBB tree. Rebuilding is necessary if the geometry is modified by calling col_add_triangle(). The interface currently does not have access to the triangles of an object at a level low enough to morph or distort portions of the object; if this is necessary, the browser must call col_clear_object() and rebuild it from scratch.

Once the objects are represented in the collision detection library's data structures, the browser calls col_update_transform() to update the position and rotation of any object, then col_test() to perform collision detection. If collision detection is only being used for navigation, as required by the specification, the browser only needs to check the return value of col_report_collision() to determine if the movement in that frame occurs without running into other geometry. If collision detection is being used for more complex applications, the browser will have to generate *collideTime* and *collideObjects* events. To do this, the browser needs to get the set of reports generated by col_report_collision() to figure out which pairs of CollisionObjects actually collided.

Automated handling of LOD or Switch nodes seems to require pushing scene graph semantics down into the collision detection routines, at the cost of code complexity. The future work section sketches possible extensions to the interface for this purpose. A method under the current interface lets the browser "manually" control these nodes:

col_open	initialize collision detection library
col_create_object	add a collidable object
col_add_triangle	add a triangle to an object
col_finish_object	build the OBB hierarchy for an object
col_clear_object	destroy an object's geometry
col_delete_object	delete a collidable object
col_activate	turn on collision detection for an object
col_deactivate	turn off all collision detection for an object
col_activate_pair	turn on collision detection between two objects
col_deactivate_pair	turn off collision detection between two objects
col_update_transform	transform (rigidly) an object
col_test	perform collision detection
col_report_collision	report collisions

Table 1: *Collision Detection Library Interface*

make a separate col_add_object() call for each child, and use col_deactivate() and col_activate() to notify the collision detection library when the Switch's *whichChoice* field changes or the LOD moves into a new range. This means the browser is responsible for synchronizing the pairwise activation of *all* the children of the node. For LOD nodes, an alternate approach exists: add only the most complex of the children to the collision detection scheme. This will only be a computational overhead when a collision or near-miss occurs, and will be *necessary* in physical simulations, where distant objects use a simple geometry for rendering but must use fully detailed geometry for their interactions with other objects.

Every time the VRML scene graph causes the browser to generate a *collide_changed* event, the browser will need to call col_activate() or col_deactivate() for the appropriate objects. Finer control of collision checking may be necessary for more advanced uses. Our collisionObject allows entities in the scene graph to precisely control which objects are checked for collision using col_activate_pair() and col_deactivate_pair().

The browser also needs to take action when a new NavigationInfo node is bound or an *avatarSize_changed* event is generated by the currently bound NavigationInfo. The object in the collision detection database that represents the user's viewpoint may need to be scaled according to the *avatarSize* field's first value. This can be done by deleting the object's geometry and rebuilding it at the proper size.

4.2 Internals

V-COLLIDE unifies the framework of the I-COLLIDE and RAPID systems. The n-body "Sweep and Prune" algorithm for filtering collisions among large numbers of objects sits at the top level of the collision detection routines, with an oriented bounding box (OBB) hierarchy providing pairwise exact contact determination for objects underneath. I-COLLIDE's pairwise collision test depends on the Lin-Canny algorithm for exact collision detection, which is correct only for convex objects or union(s) of convex pieces. Decomposing arbitrary models into convex pieces is difficult; users of packages like I-COLLIDE have had to do this as a preprocess in the past [9], but this decomposition is impractical to perform in real-time. RAPID assumes inputs are triangulated polygonal models. Rather than extending RAPID's OBB hierarchy to handle VRML's cones, spheres, and cylinders, we require the library caller to tessellate the solids within the (user-specified) tolerance desired for collision detection. It would be possible to extend V-COLLIDE to handle curved surfaces exactly. However, this could significantly increase system complexity, as we would need accurate and robust pairwise tests among all primitive types – an $O(n^2)$ problem. Our goal has been to keep the system architecture simple and elegant.

4.3 System Performance

There are three concerns when adding an extension to VRML: rendering speed, memory requirements, and implementation complexity. When no near-collisions occur, the cost is merely traversals of three lists linear in the CollisionObject complexity of the scene.

Since we did not have source-code access to any VRML 2.0 browsers, it was impossible to perform a test integration of our library. We have tested the performance of V-COLLIDE with a stand-alone, multi-body simulation. Several polygonal bunny rabbits bounce around inside a cubical volume, with simplistic rules to determine how they react to collisions with each other and the walls of the simulation volume. Graph 1 shows how the average frame time for the simulation is affected by increasing the number of bunny models in the simulation, keeping all other parameters fixed (including a measure of the "density" of the simulation). At 50 bunnies, one frame worth of collision detec-

```
PROTO CollisionObject
  [ eventIn MFNode addChildren
    eventIn MFNode removeChildren
    eventIn MFNode addIgnoredObjects
    eventIn MFNode removeIgnoredObjects
    exposedField MFNode children []
    exposedField MFNode ignoredObjects []
    exposedField SFBool collide TRUE
    field SFVec3f bboxCenter 0 0 0
    field SFVec3f bboxSize -1 -1 -1
    field SFNode proxy NULL
    eventOut SFTime collideTime
    eventOut MFNode collideObjects ] {

  Collision {
    addChildren IS addChildren
    removeChildren IS removeChildren
    children IS children
    collide IS collide
    bboxCenter IS bboxCenter
    bboxSize IS bboxSize
    proxy IS proxy
    collideTime IS collideTime
  }

  Group {
    children IS collideObjects
  }

  Group {
    addChildren IS addIgnoredObjects
    removeChildren IS removeIgnoredObjects
    children IS ignoredObjects
  }
}
```

Figure 3: *PROTO for CollisionObject, stored at http://www.cs.unc.edu/~geom/V_COLLIDE/CollisionObject.wrl*

```
DEF FOO CollisionObject {
  ...
  ignoredObjects [
    DEF BAR CollisionObject {
      ...
      ignored Objects [ USE FOO ]
    }
  ]
}
USE BAR
```

Figure 4: *Declaring a pair of CollisionObjects which ignore one another*

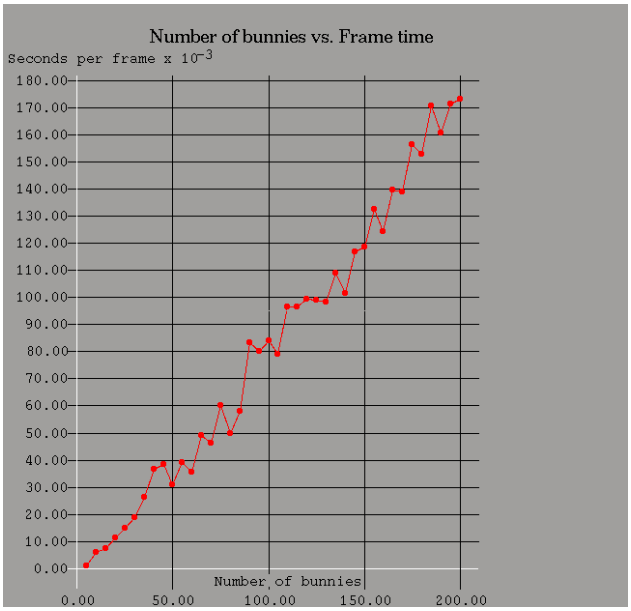

Figure 5: *System performance: linear in number of collisions*

tion requires 33 miliseconds (Figure 5). For this graph, we ran our timing tests on an HP 735/125. Each bunny is comprised of 575 triangular faces, meaning our 50-bunny test contains 28,750 polygons – more than one would expect in a typical VRML scene. The bunnies move at velocities fast enough to make the test meaningful.

Our prototype of V-COLLIDE uses considerable amounts of memory. There are two components to the memory cost: the n-body algorithm, and the pairwise algorithm. For n-body collision detection, we have an array of n^2 entries that contains between 4 and 30 bytes per entry (depending on whether collision detection for the pair of objects represented by that array entry is active or inactive), and 118 bytes per object. The pairwise algorithm currently costs nearly 400 bytes per triangle in the model.

Both of these costs can be reduced. A space-optimized version of the n-body code has the same 118 bytes per object, a sparse array requiring between $8n$ and $4n^2$ bytes, and only a further 21 bytes per *active, overlapping* (approximately colliding) pair of objects. The OBB code can be made to cost roughly 100 bytes per triangle. These economies should cause little performance degredation for the n-body algorithm, and a worst-case 25% slowdown during the pairwise tests.

The V-COLLIDE prototype grafts together two pieces of library code; both pieces are well documented and well-tested by public release to the several user communities. The interface has been simplified to match the needs of a VRML browser, so integration should require no knowledge of the underlying code. Some complexity will necessarily be added to a browser interfacing with CollisionObject nodes, but this should be localized.

4.4 Optimization

The database will sometimes contain CollisionObjects that are extremely large. Large bounding boxes containing much empty space unnecessarily degrade the performance of the n-body algorithm, since they increase the number of pairwise tests that will be performed. To avoid this, a browser can attempt to automatically decompose a CollisionObject - or geometry that has not been tagged as belonging to a particular CollisionObject - into multiple subobjects for collision purposes. We can prevent these subobjects from being meaninglessly tested against one another using col_deactivate_pair(). To decompose geometry into CollisionObjects, it is sufficient to note that two Shape nodes which are not separated by a Transform's parent are rigid with respect to one another, and so cannot collide with one another. Thus, as long as each Transform node belongs to a different CollisionObject than its siblings, correct collision detection can easily be maintained.

5 Future Work

A modification of the pairwise collision test permits us to obtain a conservative distance measure between two objects – at some cost to the speed of the test. This would enable us to provide a good approximated distance metric that might be more useful in some situations than the simple approximations commonly used today, such as the distance between objects' bounding box centers. (See Table 2)

The underlying OBB tree implementation assumes triangulated polygonal models; to handle VRML's spheres, cones, and cylinders exactly requires extensions to the code and API. Reasonable algorithms from the literature perform accurate and efficient collision between these primitives. However, adding these extensions would complicate the exact primitive intersection test module. Alternately, we could allow the user to specify the error bound and do a triangulation inside our library, taking advantage of the known structure of the collision detection algorithm to reduce the necessary number of tests.

The time required to build a reasonable OBB hierarchy is not negligible. We currently pause briefly on loading a model in order to build its hierarchy; this pause is substantial when model sizes approach the hundreds of thousands of polygons. (On a 250 MHz R4400 CPU, we need 1.5 seconds to build a tree of 45000 polygons.) Our work would benefit greatly from faster methods for building these trees well. It would also be useful to make the data structures of OBB trees more adaptable, allowing us to more easily handle changing geometry.

LOD and Switch nodes need to have their child-selection behavior reported to the collision detection library. It would be possible to move some of this intelligence down into the library, in which case the browser would need a path to tell the collision detection routines which of the node's children is currently active, so that it knows which node to test against. Unfortunately we would be evolving toward maintaining a copy of the entire scene graph within the collision detection library.

For each collision, V_COLLIDE reports only the objects involved. It would be possible to specify which faces of which geometry nodes were involved in the collision, but this would increase the memory footprint of our data structures and would require a more densely scripted environment.

With distributed simulation or multiuser use of VRML, distributed protocols come into play [1, 10, 11]. In any situation without a globally consistent state at all browsers, collision detection, like any other object-object interaction mechanism, will have to be given a great deal of thought. Most schemes proposed for seamlessly distributing VRML involve a spatial partition. The collision detection system can be extended to monitor only the local environment, thus increasing the scalability of simulations.

References

[1] H. Z. Andersson. VRML Behaviour - a proposal. http://www.lysator.liu.se/~zap/vr_prop1.html.

[2] G. Bell, R. Carey, and C. Marrin. The Virtual Reality Modeling Language Specification Version 2.0. http://vag.vrml.org/VRML2.0/FINAL/, August 1996.

[3] S. Cameron. Approximation Hierarchies and S-bounds. In *Proceedings. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, 1991.

[4] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. In *Proc. of ACM Interactive 3D Graphics Conference*, pages 189–196, 1995.

[5] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. In *Proc. of ACM Siggraph'96*, pages 171–180, 1996.

[6] M. Held, J.T. Klosowski, and J.S.B. Mitchell. Evaluation of collision detection methods for virtual reality fly-throughs. In *Canadian Conference on Computational Geometry*, 1995.

[7] M.C. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.

col_est_distance	lower bound distance between two objects
col_set_tolerance	set tolerance for tessellation of conics
col_add_sphere	add an approximate sphere to an object
col_add_cylinder	add an approximate cylinder to an object
col_add_cone	add an approximate cone to an object
col_create_switch	create a Switch or LOD
col_parent_object	set an object to be the child of a Switch or LOD
col_choose_child	specify child of Switch/LOD to use currently

Table 2: *Some Possible Extensions to Collision Detection Library Interface*

[8] Tom Meyer and D. B. Conner. Adding Behavior to VRML. *Symposium on the Virtual Reality Modeling Language*, pages 45–51, 1995.

[9] John Nagle. Personal Communication, 29 October 1996.

[10] B. Roehl. Distributed Virtual Reality – An Overview. http://sunee.uwaterloo.ca/~broehl/distrib.html. June 1995.

[11] B. Roehl. Some Thoughts on Behavior in VR Systems. http://sunee.uwaterloo.ca/~broehl/behav.html. August 1995.

[12] IETF Work-in-progress. Universal Resource Name. http://services.bunyip.com:8000/research/ietf/urn-ietf.

[13] D. Zeltzer. Autonomy, Interaction and Presence. *Presence*, 1(1):127, 1992.